Science of Computer Programming 3 (1983) 119–149
North-Holland

A HIGHER-ORDER IMPLEMENTATION OF REWRITING

Lawrence PAULSON
Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, United Kingdom

Communicated by M. Sintzoff
Received April 1983
Revised June 1983

Abstract. Many automatic theorem-provers rely on rewriting. Using theorems as rewrite rules helps to simplify the subgoals that arise during a proof.

LCF is an interactive theorem-prover intended for reasoning about computation. Its implementation of rewriting is presented in detail. LCF provides a family of rewriting functions, and operators to combine them. A succession of functions is described, from pattern matching primitives to the rewriting tool that performs most inferences in LCF proofs.

The design is highly modular. Each function performs a basic, specific task, such as recognizing a certain form of tautology. Each operator implements one method of building a rewriting function from simpler ones. These pieces can be put together in numerous ways, yielding a variety of rewriting strategies.

The approach involves programming with higher-order functions. Rewriting functions are data values, produced by computation on other rewriting functions. The code is in daily use at Cambridge, demonstrating the practical use of functional programming.

1. Introduction to rewriting

When trying to prove a theorem, one approach is to simplify it by applying left-to-right *rewrite rules*. For example, consider the proof that addition of natural numbers is associative. We can take the natural numbers to have the form 0, SUCC(0), SUCC(SUCC(0)), ..., and define addition using the axioms

$$0+n=n,$$

$$\mathrm{SUCC}(m)+n=\mathrm{SUCC}(m+n).$$

Associativity of addition can be stated

$$(m+n)+k=m+(n+k).$$

Induction on m reduces the problem to proving the two goals:

$$(0+n)+k=0+(n+k) \quad \text{(base case)}$$

$$(\mathrm{SUCC}(m)+n)+k=\mathrm{SUCC}(m)+(n+k) \quad \text{(step case)}$$

with induction hypothesis $(m+n)+k=m+(n+k)$.

When conducting such a proof by hand, we would not even write down an expression like $0+(n+k)$, but would cross out the 0 immediately, proving the base case. For the step case, we would expand out both sides, writing

- left side:

$$\begin{aligned}&(\text{SUCC}(m)+n)+k && \text{(by definition of addition)}\\ &\quad=\text{SUCC}(m+n)+k && \text{(by definition of addition)}\\ &\quad=\text{SUCC}((m+n)+k) && \text{(by induction hypothesis)}\\ &\quad=\text{SUCC}(m+(n+k))\end{aligned}$$

- right side:

$$\begin{aligned}&\text{SUCC}(m)+(n+k) && \text{(by definition of addition)}\\ &\quad=\text{SUCC}(m+(n+k)).\end{aligned}$$

Note that the cancelling of the 0 in $0+(n+k)$ can be regarded as the *rewriting* of $0+(n+k)$, using the first axiom of addition as a rewrite rule. The other addition axiom, and the induction hypothesis, are also used as rewrite rules in the proof.

Rewriting is fundamental to most research on proving theorems by computer [2,6,13]. Many theorems can be proved by induction and rewriting, or even by rewriting alone. For theories where all axioms are equations, it is sometimes possible to prove additional equations and achieve a *complete set* of rewrite rules, capable of rewriting any term into canonical form [9]. This provides a decision procedure for testing the validity of *any* equation of the theory.

There is a wide variety of rewriting strategies, as Kuechlin [10] thoroughly discusses. In what order should rewrite rules be considered if more than one applies? What about *looping* rewrites such as $m+n=n+m$? Should a term be rewritten again and again until no rules apply, or only a bounded number of times? Should a term be traversed top-down or bottom-up, left-to-right or right-to-left? And how can we make use of rewrites that have pre-conditions, such as

if m is not zero **then** $(m*n)/m=n$?

This paper presents a family of rewriting primitives, and operators to combine them. It proceeds in stages, from pattern-matching primitives, to instantiation functions, term and formula rewriting functions, tautology solvers, finally discussing the rewriting theorem-prover that is provided in the LCF system [13]. The functions at each stage are constructed from those of the previous stage by functional operators that express simple computational intuitions such as sequencing, alternation, and repetition.

Most function definitions are brief and directly express the design decisions such as traversal order. Even the theorem-proving function is only twelve lines, which form a readable summary of its rewriting strategy. If you dislike the standard strategy, you can easily implement another. This modularity is in sharp contrast to large, monolithic implementations of rewriting, such as LCF's previous one.

Higher-order functions play a vital role. The rewriting primitives are functions, the operators are functionals. The method is implemented in the programming

language ML [7, 8]. It cannot be implemented in Backus's FP language [1], which allows only first-order functions.

2. The LCF proof assistant

LCF is an interactive theorem-prover for Dana Scott's Logic for Computable Functions [7]. This logic, called PPLAMBDA, provides the 'undefined' element $\perp$ and the associated partial ordering; it allows reasoning about denotational semantics, programs that manipulate unbounded streams, etc. LCF lets you declare constants and types, assert axioms, and prove theorems; it records all this information in a theory file. You can build a theory on top of other theories, inheriting the constants, axioms, etc. of the other theories. A major verification project might involve a theory hierarchy containing theories of lists, natural numbers, sets, specialized data structures, and theories of increasingly complex functions operating on these types.

LCF lies mid-way between a step-by-step proof checker, and an automatic theorem-prover. When conducting a proof, you are responsible for performing each logical inference. You may automate parts of this task by writing programs in the *meta-language*, ML. The standard theorem-proving tools, such as the rewriting functions, are written in ML. ML treats the terms and formulas of PPLAMBDA as data values, providing functions to build them and take them apart. Theorems are also data values.

PPLAMBDA is a *natural deduction* logic [12]: theorems are proved relative to a set of assumptions. A theorem $[A_1; \ldots; A_n] \vdash B$ means that the *conclusion* B holds whenever the *assumptions* $A_1, \ldots, A_n$ hold. (The conclusion and assumptions are formulas.) In ML, a theorem is a value of the abstract type *thm*, represented by a pair $([A_1; \ldots; A_n], B)$. Such a pair is traditionally called a *sequent*.

ML provides functions to decompose a theorem into its conclusion and assumptions, but not to construct an arbitrary theorem. Theorems must be *proved* by applying inference rules to axioms. LCF uses typical introduction/elimination inference rules, implemented as functions whose arguments and results have type *thm*. Type-checking guarantees that inference rules are only applied to theorems.

Most proofs are conducted *backwards*, starting from the desired goal (expressed as a sequent). The goal is reduced to simpler subgoals, and those reduced to further subgoals, until all the subgoals have been reduced to tautologies. Backwards proof uses *tactics*. A tactic is a function that maps a goal to a list of subgoals, paired with an inference rule. The inference rule justifies the choice of subgoals. Given theorems that assert the subgoals, it produces a theorem that asserts the goal.

Tactics correspond to simple reasoning methods. For instance, the conjunction tactic CONJ_TAC reduces any goal of the form $A \wedge B$ to the two subgoals A and B; the discharge tactic DISCH_TAC reduces any goal of the form $A \rightarrow B$ to the subgoal of proving B under the assumption A. If you apply a tactic to a goal that

it cannot handle, then it will *fail* via ML's exception mechanism (described below).

It would be tedious indeed to prove theorems in such tiny steps as CONJ_TAC and DISCH_TAC. Most interactive proofs rely on more powerful tactics, constructed from the primitive ones using functions called *tacticals*. The basic ones are THEN, ORELSE, and REPEAT:

- tac_1 THEN tac_2
  applies tac_1, then applies tac_2 to all resulting subgoals;
- tac_1 ORELSE tac_2
  applies tac_1, if it fails then applies tac_2;
- REPEAT *tac*
  applies *tac* recursively on the goal and the resulting subgoals, returning the subgoals for which *tac* fails.

You may be surprised to see the imperative notions of sequencing, alternation, and repetition embodied in higher-order functions that manipulate proof strategies. This paper will show that the same notions apply to rewriting.

3. Reference summary of ML

This section describes just enough of ML to enable you to follow the rest of the paper. For instance, ML's polymorphic type system is not discussed because most of the functions in this paper are strongly typed. Gordon [8] gives a good introduction to ML and its use in theorem-proving. The Edinburgh LCF Manual [7] also contains an introduction to ML, and is helpful to have at hand if you intend to study the paper in detail.

Note: this paper concerns Cambridge LCF, a revised version of Edinburgh LCF. Most of the changes involve the logic PPLAMBDA [14]. Despite incompatibilities, documentation on Edinburgh LCF is still useful.

3.1. Values

Values of the language ML include

- the *integers* 0, 1, 2, ..., with operators +, −, *, /, etc.;
- the *booleans true* and *false*, with operators &, or, not, along with the conditional expression: **if** *b* **then** *x* **else** *y*;
- *tokens*, which are character strings such as 'NO_CONV' or 'Hi Daddy';
- *pairs* such as 0, *true*—which may be iterated to form *tuples*: 'manny', 'moe', 'jack';
- *lists* $[x_1; \ldots; x_n]$, such as [(0, *true*); 1, *false*)];
- *functions*, which may be passed as values, retaining their original variable bindings even outside of the scope where they were created;
- *terms*, *formulas*, *theorems* of the logic PPLAMBDA.

3.2. Types

Types include

int	(integers)
bool	(booleans)
tok	(tokens)
$\alpha \# \beta$	(pairs with components of the types α and β)
α *list*	(lists with elements of type α)
$\alpha \rightarrow \beta$	(functions from α to β)
term	(PPLAMBDA terms)
form	(PPLAMBDA formulas)
thm	(PPLAMBDA theorems).

3.3. Declarations

Typed at top level, the declaration

let $x = 3$;;

causes x to denote 3 for the rest of the terminal session. Inside a program, the declaration

let $x = 3$ **in** $x + x$

causes x to denote 3 only in the expression $x + x$. A local declaration supersedes any global one, using static scope rules.

Declarations include

let $x = 3$	(values)
let *double* $k = k + k$	(functions)
let *times* $x\ y = x * y$	(curried functions)
letrec *fact* $n =$	(recursive functions)
if $n = 0$ **then** 1	
else *times* n (*fact* $(n - 1)$).	

Note that parentheses are not required around arguments to functions. Thus *double* k means the same as *double*(k).

3.4. Trapping failures

An attempt to perform an illegal operation, such as division by zero, causes ML to signal *failure*. Each failure has an associated error message, or *failure token*. A program can signal failure via the expression

failwith 'my error'.

A failure halts execution unless it is *trapped*. Evaluating the expression $x?y$ computes the value of x; if x fails, then it computes the value of y.

Some people object to failure trapping, comparing it to a goto statement. They prefer to use conditional expressions to prevent failures from occurring. This is impractical for combining proof strategies, where success or failure is difficult to predict. Though implemented as a goto, failure can be understood as an error value that is passed along until it is tested for. A common use of failure is to reject inputs that do not match an expected pattern.

3.5. Standard functions and operators

Infix operators include

- $\cdot$ —'conses' an element to the front of a list

$$x \cdot [y_1; \ldots; y_n] \rightarrow [x; y_1; \ldots; y_n];$$

- @ —appends two lists

$$[x_1; \ldots; x_m] @ [y_1; \ldots; y_n] \rightarrow [x_1; \ldots; x_m; y_1; \ldots; y_n];$$

- $\circ$ —composes two functions

$$(f \circ g)x \rightarrow f(g\,x).$$

Functions include

- *map* —applies a function to each element of a list

$$map\,f[x_1; \ldots; x_n] \rightarrow [f\,x_1; \ldots; f\,x_n];$$

- *mapfilter* —like *map* but does not propagate failure; ignores any list elements for which f fails;

- *flat* —flattens a list of lists into a list

$$flat\,[l_1; \ldots; l_n] \rightarrow l_1 @ \cdots @ l_n;$$

- *itlist* —iterates down a list, accumulating a result

$$itlist\,f[x_1; \ldots; x_n]\,y \rightarrow f\,x_1\,(f \ldots (f\,x_n\,y));$$

- *fst, snd* —select components of pairs

$$fst(x, y) \rightarrow x,$$

$$snd(x, y) \rightarrow y.$$

3.6. Manipulating PPLAMBDA abstract syntax

The terms of PPLAMBDA, which denote computable values, are elements of the ML type *term*. Terms have four abstract suntax classes:

C	(constant, where C is a constant symbol)
x	(variable)

$\lambda x\,.\,t$ (lambda-abstraction)
$t\,u$ (combination, or function application).

There is syntactic sugar for certain common expressions:

$p \Rightarrow t \,|\, u$ (conditional expression, actually "COND $p\,t\,u$")
t, u (pair, actually "PAIR $t\,u$")

You may declare constant symbols; the standard ones include

$\bot$	(bottom, or undefined element)
TT	(true, a computable truth value)
FF	(false)
FST	(a selector function, like *fst* in ML)
SND	(like *snd* in ML).

Note: though not discussed here, PPLAMBDA terms obey a polymorphic type system similar to ML's.

The formulas of PPLAMBDA, which denote logical statements, are elements of the ML type *form*. Formulas have seven abstract syntax classes:

$\forall x.A$	(universal quantifier)
$\exists x.A$	(existential quantifier)
$A \wedge B$	(conjunction)
$A \vee B$	(disjunction)
$A \rightarrow B$	(implication)
$A \leftrightarrow B$	(if-and-only-if)
$P\,t$	(predicate, where P is a predicate symbol).

Standard formulas include

TRUTH( )	(standard tautology predicate)
FALSITY( )	(standard contradiction predicate)
$t \equiv u$	(equivalence of t and u, actually "*equiv*(t, u)")
$t \sqsubseteq u$	(Scott inequivalence of t and u, actually "*inequiv*(t, u)")
$\neg A$	(negation, actually "$A \rightarrow$ FALSITY( )").

Note: the formula $t \equiv u$ expresses *equivalence* rather than *equality*. Roughly speaking, t and u are equivalent if they are both undefined, or both defined and equal.

The ML expression for constructing a PPLAMBDA object (term or formula) consists of that object enclosed in quotation marks. These are syntax trees, not character strings! ML provides destructor functions for taking apart PPLAMBDA objects:

dest_comb "$f\,x$" → "f", "x"
dest_equiv "$t \equiv u$" → "t", "u"
dest_conj "$A \wedge B$" → "A", "B"
dest_imp "$A \rightarrow B$" → "A", "B"
etc.

The theorems of PPLAMBDA, which denote proved formulas, are elements of the ML type *thm*. Inference rules are provided as functions that produce theorems. For instance, the function MP implements Modus Ponens.

$$\text{MP}\text{“}\vdash A \to B\text{”}\ \text{“}\vdash A\text{”} \to \text{“}\vdash B\text{”}.$$

Note: this example, and others below, show theorems in quotation marks. These depict values rather than expressions. ML does not allow quoted theorems as expressions, since they would construct theorems without proof. Inference rules like MP must be applied to identifiers or other expressions of type *thm*.

The function *concl* returns the conclusion of a theorem:

$$\mathit{concl}\ \text{“}[A_1; \ldots; A_n]\vdash B\text{”} \to \text{“}B\text{”}.$$

4. Pattern matching primitives

Now we are ready to examine LCF's rewriting functions, starting with primitives and working upwards. The ML programs below omit most PPLAMBDA inferences, as well as code for optimization and debugging. The most frequently used inferences are 'wired in' as additional primitives, to avoid deriving them repeatedly. However, the programs running in LCF are essentially as described here.

A quantified theorem such as $\vdash \forall x.A$ stands for an infinity of theorems, one for each x. In a proof, you are likely to need some of these instances, rater than the general form. LCF's pattern matching primitives relieve you of the tedium of instantiating theorems.

4.1. Matching terms and formulas

Most rewriting functions depend on the matching functions *term_match* and *form_match*. If *pattern* and *object* are terms, the call

term_match pattern object

returns a list of (*term*, *variable*) pairs. This expresses the object as an *instance* of the pattern (allowing for renaming of bound variables). If no match exists, *term_match* fails. The analogous function for formulas is *form_match*.

Let us look at a terminal session using these functions. The lines beginning with # denote input to LCF, and the other lines denote the response.

We bind a complex term, a conditional, to the ML identifier *tm_obj*. ML responds by printing the value and its type.

```
#let tm_obj = "TT⇒(FF, TT)|(TT, FF)";;
tm_obj = "(TT⇒(FF, TT)|(TT, FF))" : term.
```

A variable can be matched to the term. (The resulting match includes a list for PPLAMBDA types such as ":*", which we ignore for simplicity.)

```
#term_match "x" tm_obj;;
["(TT ⇒ (FF, TT) | (TT, FF))", "x"],
[":tr # tr", ":*"]
:((term # term) list # (type # type) list).
```

The term can be matched against a conditional, breaking it into parts:

```
#term_match "p ⇒ x | y" tm_obj;;
["TT, FF", "y"; "FF, TT", "x"; "TT", "p"],
[":tr # tr", ":*"]
:((term # term) list # (type # type) list).
```

Since "$p \Rightarrow x \mid y$" is merely syntactic sugar for "COND $p\ x\ y$", we can also match the term against a combination:

```
11028  #term_match "f x" tm_obj;;
       ["TT, FF", "x"; "COND TT (FF, TT)", "f"],
       [":tr # tr", ":**"; ":tr # tr", ":*"]
       :((term # term) list # (type # type) list).
```

We cannot match the term against something of a different form. If we try, *term_match* fails:

```
#term_match "p ⇒ FF | y" tm_obj;;
evaluation failed  term_match

#term_match "λp.p" tm_obj;;
evaluation failed  term_match.
```

We can conduct a similar session using *form_match*:

```
#let fm_obj = "∀x. (x, TT) ≡ ⊥";;
fm_obj = "∀x. (x, TT) ≡ ⊥" : form

#form_match "∀y. (x, y) ≡ ⊥" fm_obj;;
evaluation failed  form_match

#form_match "∃x. (x, TT) ≡ ⊥" fm_obj;;
evaluation failed  form_match

#form_match "∀y. (y, z) ≡ ⊥" fm_obj;;
["TT", "z"],
[":tr", ":**"]

:[[term # term) list # (type # type) list).
```

4.2. *Instantiating theorems by matching*

The functions *term_match* and *form_match* are too primitive for most applications. Their main purpose is to implement the next level of abstraction, which provides functions for instantiating theorems. Calling

PART_TMATCH *partfn A t*

matches the term t to some part of the theorem A obtained by the function *partfn*, and returns A with its types and variables instantiated. The *partfn* is composed from destructor functions such as *fst*, *dest_comb*, and *dest_equiv*. It is applied to the conclusion of the theorem after removing outer universal quantifiers.

PART_TMATCH makes it easy to define inference rules that involve matching. For instance, PPLAMBDA includes an axiom stating that the bottom element is smaller than any other element.

MINIMAL $\vdash \forall x.\ \bot \sqsubseteq x.$

Instantiating MINIMAL is inconvenient (not only x, but PPLAMBDA types must be instantiated), so LCF provides an inference rule MIN that maps any term t to the theorem $\vdash \bot \sqsubseteq t$. This rule can be expressed using PART_TMATCH and the function (*snd* ∘ *dest_inequiv*):

let MIN = PART_TMATCH (*snd* ∘ *dest_inequiv*) MINIMAL;;

Let us see how PART_TMATCH determines what part of MINIMAL to match against. This computation takes place before a term t is applied, since PART_TMATCH is a curried function:

(*snd* ∘ *dest_inequiv*) (*concl* "$\vdash \bot \sqsubseteq x$")
snd (*dest_inequiv* "$\bot \sqsubseteq x$")
snd ("$\bot$", "x")
"x".

So MIN matches a term t against the right-hand side, x, returning the instance $\vdash \bot \sqsubseteq t$ of MINIMAL.

The function PART_FMATCH is analogous, but matches some subformula of the theorem rather than a subterm. One of its applications is a simple resolution rule, MATCH_MP. This is a Modus Ponens that matches an implication to an antecedent:

```
let MATCH_MP impth =
    let match = PART_FMATCH (fst ∘ dest_imp) impth
    in
    λth. MP (match (concl th)) th;;
```

Calling MATCH_MP "$\vdash \forall x_1 \ldots x_n. A \to B$" "$\vdash A'$", where A' is an instance of A, returns the corresponding instance of B. By convention, we write this instance $\vdash B'$. The composite function (*fst* ∘ *dest_imp*) takes the first part of an implication, which is the antecedent. One of my proofs [13] involves a total function VARS_OF and a theory of strict lists. The function MAP, which maps any function f over a list, produces a total function if f is total. The rule MATCH_MP can prove that the function (MAP VARS_OF) is total.

First we load, from theory files, two theorems about totality:

```
#let VARS_OF_TOTAL = theorem 'VARS_OF' 'VARS_OF_TOTAL';;
VARS_OF_TOTAL = "⊢∀t. ¬t ≡ ⊥ → ¬VARS_OF t ≡ ⊥" : thm

#let MAP_TOTAL = theorem 'list_fun' 'MAP_TOTAL';;
MAP_TOTAL =
"⊢∀f.
   (∀x. ¬x ≡ ⊥ → ¬f x ≡ ⊥) →
   (∀l. ¬l ≡ ⊥ → ¬MAP f l ≡ ⊥)"
: thm.
```

Using MATCH_MP, we generate more totality theorems:

```
#let TOTAL1 = MATCH_MP MAP_TOTAL VARS_OF_TOTAL;;
TOTAL1 = "⊢∀l.¬l ≡ ⊥ → ¬MAP VARS_OF l ≡ ⊥" : thm

#let TOTAL2 = MATCH_MP MAP_TOTAL TOTAL1;;
TOTAL2 = "⊢∀l. ¬l ≡ ⊥ → ¬MAP(MAP VARS_OF)l ≡ ⊥" : thm

#let TOTAL3 = MATCH_MP MAP_TOTAL TOTAL2;;
TOTAL3 = "⊢∀l. ¬l ≡ ⊥ → ¬MAP(MAP(MAP VARS_OF))l ≡ ⊥" : thm.
```

5. Term conversions

The purpose of rewriting is to convert any term t into a term u that is somehow simpler. Furthermore, u must be *proved equivalent* to t. Most theorem-provers take for granted that their rewriting functions are reliable, but the LCF methodology demands that every rewriting step be justified by a theorem.

LCF's rewriting functions are called *conversions*. A term conversion is any function that maps a term t to a theorem $\vdash t \equiv u$. This converts the term t to another term u, and proves the two equivalent. Since ML allows us to take theorems apart, we can extract the new term u from the theorem $\vdash t \equiv u$.

Let us bind some ML identifiers to terms for use in later terminal sessions. (This simultaneous declaration resembles Lisp's 'destructuring let'.)

```
#let [abs1; abs2; condu; condt; condf; condfst] = example_terms;;
abs1 = "(λfun.(fun(TT, FF)⇒x | y))FST" : term
abs2 = "(λt.(λu.t, u)FF)TT" : term
condu = "(⊥⇒(TT, FF, p) | (q, TT, q))" : term
condt = "(TT⇒x | y)" : term
condf = "(FF⇒f x | f y)" : term
condfst = "(FST(TT, FF)⇒x | y)" : term.
```

5.1. Basic conversions

Beta conversion, BETA_CONV, is standard in LCF. If x is a variable and u, v are terms, and $u[v/x]$ denotes the substitution of v for x in u, then

BETA_CONV "$(\lambda x.u)v$" → "$\vdash(\lambda x.u)v \equiv u[v/x]$".

This session demonstrates that BETA_CONV performs exactly one beta conversion, not two or zero:

```
#BETA_CONV abs1;;
"⊢(λfun.(fun(TT,FF)⇒x|y))FST ≡ (FST(TT, FF)⇒x|y)" : thm

#BETA_CONV abs2;;
"⊢(λt.(λu.t, u)FF)TT ≡ (λu.TT, u)FF" : thm

#BETA_CONV condt;;
evaluation failed  BETA_CONV.
```

Another basic conversion is to rewrite according to a theorem that states an equivalence. Such theorems are called *rewrites* or *term rewrites*:

REWRITE_CONV "$\vdash\forall x_1 \ldots x_n.t \equiv u$".

This conversion takes any instance of t, such as t', and instantiates the variables $x_1 \ldots x_n$, to return the theorem $\vdash t' \equiv u'$. It is implemented using the matching function PART_TMATCH:

```
let REWRITE_CONV = PART_TMATCH (fst ∘ dest_equiv);;
```

Let us explore REWRITE_CONV. First we bind some PPLAMBDA axioms to identifiers, for use in later sessions.

```
#let [COND_⊥; COND_TT: COND_FF;
#     MIN_COMB: MIN_ABS;
#     MK_PAIR; FST_PAIR; SND_PAIR]
# = example_rewrites;;
COND_⊥ = "⊢(⊥⇒x|y) ≡ ⊥" : thm
COND_TT = "⊢(TT⇒x|y) ≡ x" : thm
COND_FF = "⊢(FF⇒x|y) ≡ y" : thm
MIN_COMB = "⊢⊥x ≡ ⊥" : thm
MIN_ABS = "⊢λx.⊥ ≡ ⊥" : thm
MK_PAIR = "⊢FST x, SND x ≡ x" : thm
FST_PAIR = "⊢FST(x, y) ≡ x" : thm
SND_PAIR = "⊢SND(x, y) ≡ y" : thm.
```

We build a conversion from FST_PAIR. It can simplify terms beginning with FST, but not those beginning with SND:

```
#let FST_CONV = REWRITE_CONV FST_PAIR;;
FST_CONV = - : conv

# FST_CONV "FST(TT, FF)";;
"⊢FST(TT, FF) ≡ TT" : thm

# FST_CONV "SND(TT, FF)";;
evaluation failed  term_match.
```

We build a conversion from COND_TT and use it to simplify a term:

```
#let COND_TT_CONV = REWRITE_CONV COND_TT;;
COND_TT_CONV = - : conv

# COND_TT_CONV condt;;
"⊢(TT⇒x | y) ≡ x" : thm.
```

This conversion insists that the condition be TT, not noticing that FST(TT, FF) has the same value:

```
# COND_TT_CONV condfst;;
evaluation failed  term_match.
```

5.2. *Combining conversions*

We have operators, similar to tacticals, for combining BETA_CONV and REWRITE_CONV into more powerful conversions.

The operator ORELSEC provides the notion of *alternation*. For conversions $conv_1$ and $conv_2$, and term t, the conversion

$$(conv_1 \text{ ORELSEC } conv_2)t \;\rightarrow\; conv_1\ t ?conv_2\ t.$$

It tries $conv_1$; if that fails, then it tries $conv_2$.

Using ORELSEC, we can implement a conversion for conditionals that handles both TT and FF, though still not FST(TT, FF):

```
#let COND_TF_CONV =
#    (REWRITE_CONV COND_TT) ORELSEC
#    (REWRITE_CONV COND_FF);;
COND_TF_CONV = - : conv

# COND_TF_CONV condt;;
"⊢(TT⇒x | y) ≡ x" : thm

# COND-TF_CONV condf;;
"⊢(FF⇒f x | f y) ≡ f y" : thm

# COND_TF_CONV condfst;;
evaluation failed  term_match.
```

We can also implement the notion of *sequencing*, defining an operator THENC. For conversions $conv_1$ and $conv_2$, the conversion

$(conv_1$ THENC $conv_2)\ t$

derives

$\vdash t \equiv t1$ (by $conv_1$)
$\vdash t1 \equiv t2$ (by $conv_2$)

and returns

$\vdash t \equiv t2$ (by transitivity, failing if $conv_1$ or $conv_2$ does).

Note that THENC justifies its result using the justifications produced by $conv_1$ and $conv_2$. Using THENC, we can implement a double beta-conversion, which fails if only one beta-conversion is possible:

```
#let BETA_BETA_CONV = BETA_CONV THENC BETA_CONV;;
BETA_BETA_CONV = - : conv

# BETA_BETA_CONV abs1;;
evaluation failed  BETA_CONV

# BETA_BETA_CONV abs2;;
"⊢(λt.(λu.t, u)FF)TT ≡ TT, FF" : thm.
```

Both ORELSEC and THENC have identity elements. The conversion NO_CONV applies to no terms; it always fails. The conversion ALL_CONV applies to all; it maps any term t to the theorem $\vdash t \equiv t$:

```
# NO_CONV condt;;
evaluation failed  NO_CONV

# ALL_CONV condfst;;
"⊢(FST(TT, FF) ⇒ x | y) ≡ (FST(TT, FF) ⇒ x | y)" : thm.
```

For combining several conversions into a multi-way choice, use FIRST_CONV. It is defined using itlist, ORELSEC, and NO_CONV:

```
FIRST_CONV [conv1; ...; convn] →
  conv1 ORELSEC ... ORELSEC convn.
```

Using FIRST_CONV, we can implement a conversion for conditionals that handles the conditions ⊥, FF, and TT.

```
#let COND_CONV =
#    FIRST_CONV (map REWRITE_CONV [COND_TT; COND_FF; COND_⊥]);;
COND_CONV = - : conv

# COND_CONV condu;;
"⊢(⊥ ⇒ (TT, FF, p) | (q, TT, q)) ≡ ⊥" : thm
```

```
#COND_CONV condf;;
"⊢(FF⇒f x | f y) ≡ f y" : thm

#COND_CONV condt;;
"⊢(TT⇒x | y) ≡ x" : thm

#COND_CONV condfst;;
evaluation failed  FIRST_CONV.
```

These operators resemble tacticals. THENC and THEN both express sequencing; ORELSEC and ORELSE both express alternation. The tactic ALL_TAC, which passes on its goal unchanged, is the identity for THEN. (ALL_TAC is called IDTAC in Edinburgh LCF.) The tactic NO_TAC, which fails on all goals, is the identity for ORELSE. ORELSEC and NO_CONV are implemented like ORELSE and NO_TAC, using failure. Most remarkably, we can define *repetition* for conversions exactly as it is defined for tactics:

```
letrec REPEATC conv t =
    ((conv THENC (REPEATC conv)) ORELSEC ALL_CONV)t;;
```

A fine point: Without the abstraction over *t*, REPEATC would always loop, because ML uses applicative order (eager) evaluation rather than normal order (lazy) evaluation.

Using REPEATC, we can implement a function that performs as many top-level beta-conversions as possible:

```
#let BETA_N_CONV = REPEATC BETA_CONV;;
BETA_N_CONV = - : conv

#BETA_N_CONV abs1;;
"⊢(λfun.(fun(TT, FF)⇒x | y))FST ≡ (FST(TT, FF)⇒x | y)" : thm

#BETA_N_CONV abs2;;
"⊢(λt.(λu.t,u)FF)TT ≡ TT, FF" : thm

#BETA_N_CONV condt;;
"⊢(TT⇒x | y) ≡ (TT⇒x | y)" : thm.
```

5.3. Depth conversions

Now we step beyond the analogy with tacticals, and examine conversions that traverse terms recursively. LCF provides functions for converting subterms: COMB_CONV handles combinations, while ABS_CONV handles abstractions. They fail on terms that do not have the corresponding form.

The conversion (COMB_CONV *conv* "*f t*") derives

```
⊢f ≡ g        (by conv)
⊢t ≡ u        (by conv)
```

and returns

$$\vdash f\,t \equiv g\,u \qquad \text{(by substitution).}$$

The conversion (ABS_CONV *conv* "$\lambda x.t$") derives

$$\vdash t \equiv u \qquad \text{(by } conv\text{)}$$

and returns

$$\vdash \lambda x.t \equiv \lambda x.u \quad \text{(by extensionality, possibly renaming } x\text{).}$$

Let us combine COMB_CONV and ABS_CONV into a conversion for a term's top-level subterms. Recall that a term can be a constant, variable, abstraction, or combination. Constants and variables are left unchanged, using ALL_CONV:

```
let SUB_CONV conv =
    FIRST_CONV [COMB_CONV conv; ABS_CONV conv; ALL_CONV];;
```

Now it is simple to write a conversion DEPTH_CONV that recursively rewrites all subterms of a term, in depth-first order.

```
letrec DEPTH_CONV conv t =
      (SUB_CONV (DEPTH_CONV conv) THENC (REPEATC conv))t;;
```

To try DEPTH_CONV out, we first make a top-level conversion that includes beta-conversion and all our rewrites:

```
# let MANY_CONV =
#     FIRST_CONV (map REWRITE_CONV rewrites) ORELSEC
#     BETA_CONV;;
MANY_CONV = - : conv.
```

Now we make a depth conversion from MANY_CONV and try it on some examples:

```
# let D_CONV = DEPTH_CONV MANY_CONV;;
D_CONV = - : conv

# D_CONV abs2;;
"⊢(λt.(λu.t, u)FF)TT ≡ TT, FF" : thm

# D_CONV condfst;;
"⊢(FST(TT, FF) ⇒ x | y) ≡ x" : thm

# D_CONV abs1;;
"⊢(λfun.(fun (TT, FF) ⇒ x | y))FST ≡ (FST(TT, FF) ⇒ x | y)" : thm.
```

We have finally managed to simplify *condfst*, but what happened with *abs*1? Clearly its result can be simplified further. We need a more sophisticated conversion,

which resimplifies the result of every successful conversion:

```
letrec REDEPTH_CONV conv t =
       (SUB_CONV (REDEPTH_CONV conv) THENC
        ((conv THENC (REDEPTH_CONV conv)) ORELSEC ALL_CONV))
       t;;
```

Here we see that REDEPTH_CONV can simplify *abs* 1 completely:

```
#let RD_CONV = REDEPTH_CONV MANY_CONV;;
RD_CONV = - : conv

#RD_CONV abs 1;;
"⊢(λfun.(fun(TT, FF)⇒x | y))FST≡x" : thm.
```

DEPTH_CONV and REDEPTH_CONV rewrite subterms before rewriting the top-level term. You may prefer TOP_DEPTH_CONV, which tries to rewrite the term before its subterms. This can be quicker, converting FST(x, y) to x without wasting time on y. It can also be slower, converting $(\lambda x.F\,x\,x)t$ to $(F\,t\,t)$ and then converting t twice:

```
letrec TOP_DEPTH_CONV conv t =
       (REPEATC conv THENC
        (SUB_CONV (TOP_DEPTH_CONV conv)) THENC
        ((conv THENC (TOP_DEPTH_CONV conv)) ORELSEC ALL_CONV)
       t;;
```

6. Interlude

Though we have passed over numerous programs and examples, we are only half-way through Cambridge LCF's implementation of rewriting. Let us pause and reflect on what we have seen so far.

In one sense, there is nothing remarkable about any of the programs above. Pattern matching and rewriting have been around for decades. The old simplifier in Edinburgh LCF [7] is as powerful as TOP_DEPTH_CONV, and other implementations of rewriting are considerably more elaborate [10].

My approach differs in its modular programming style, which provides both flexibility and readability. The conversions and their operators form a language for expressing rewriting strategies. For instance,

```
REPEATC (BETA_CONV THENC BETA_CONV)
```

evidently performs an even number of beta-conversions. Conversions were developed in order to escape the rigidity of the old LCF simplifier; now several LCF users are using conversions to suit their particular needs.

Each depth conversion expresses an abstract strategy for the traversal of terms, independent from the conversion to be applied at each subterm. The code for

TOP_DEPTH_CONV is a paraphrase of its effect. 'Repeatedly apply the conversion *conv* to the term t as long as possible; then convert recursively the subterms; if the result can still be converted, then convert recursively again.'

Conversions may be amenable to algebraic reasoning, since they obey certain identity, associative, and distributive laws. Consider the equations

$$0+a=a+0=a$$
$$(a+b)+c=a+(b+c)$$
$$a+b=b+a$$
$$a+a=a$$

$$1\cdot a=a\cdot 1=a$$
$$(a\cdot b)\cdot c=a\cdot(b\cdot c)$$
$$0\cdot a=a\cdot 0=0$$

$$a\cdot(b+c)=a\cdot b+a\cdot c$$
$$(b+c)\cdot a=b\cdot a+c\cdot a.$$

Researchers [11] have begun to study mathematical structures such as *semirings* and *regular algebras*, which satisfy various subsets of these equations. Putting ORELSEC, THENC, NO_CONV, ALL_CONV for $+, \cdot, 0, 1$, it appears that conversions satisfy most of them. The main exception is $a+b=b+a$; the operator ORELSEC is not symmetric, but always tries its left operand first. Also, the second distributive law fails. The forgoing also applies to tactics.

Most of the research on semirings concerns path-finding in graphs. One can imagine a graph where the nodes are terms and the arcs are conversions. Whether or not this has any practical application, it is important that conversions can be understood through mathematical theories that have arisen in unrelated branches of computer science.

Backus's Turing Award lecture [1] has attracted so much attention that many people now equate functional programming with his FP systems. Indeed, conversions exemplify 'changeable parts', 'combining forms', and 'algebra of programs', which Backus claims as advantages of his technique. However, FP systems allow only first-order programming. FP does not regard functions as data objects; for instance, you cannot build a list of functions. Conversions, and many other parts of LCF, rely heavily on higher-order functions. FP provides only a fixed set of functionals ('combining forms'); a programmer cannot introduce new ones such as TOP_DEPTH_CONV. FP does not allow the trapping of failures, which LCF requires for combining tactics and conversions. It is vital to recognize that the FP style of programming differs fundamentally from the ML style.

7. Formula conversions

Now we resume the examination of LCF's rewriting tools. The ideas behind term conversions apply equally well to the rewriting of formulas. Let a *formula conversion*

be any function that maps a formula A to a theorem $\vdash A \leftrightarrow B$. This converts A to B and proves the two equivalent. LCF provides a family of formula conversions, and operators to combine them, as for term conversions.

7.1. Analogs of term conversions

We can rewrite a formula with a theorem that states a logical equivalence, by invoking the formula conversion

REWRITE_FCONV "$\vdash \forall x_1 \ldots x_n.A \leftrightarrow B$".

Such theorems are called *formula rewrites*. The conversion is implemented like REWRITE_CONV, using the instantiation function PART_FMATCH:

let REWRITE_FCONV = PART_FMATCH (*fst* ∘ *dest_iff*);;

This is useful for expanding out the definition of a predicate, such as

$$\vdash \forall rel.\ \text{TRANSITIVE}\ rel \leftrightarrow \forall x\ y\ z.rel\ x\ y \equiv \text{TT} \wedge rel\ y\ z \equiv \text{TT} \rightarrow rel\ x\ z \equiv \text{TT}.$$

LCF provides the identity conversions NO_FCONV, which always fails, and ALL_FCONV, which maps any formula A to $\vdash A \leftrightarrow A$. For sequencing, the conversion ($fconv_1$ THENFC $fconv_2$) is defined in terms of Modus Ponens. The operators ORELSEFC, REPEATFC, and FIRST_FCONV are implemented like their term counterparts.

We can also convert subterms and subformulas. If P is a predicate, then

PRED_FCONV *conv* "$P(t)$"

is a formula conversion that converts the argument t, deriving

"$\vdash t \equiv u$"	(by *conv*)
"$\vdash P(t) \rightarrow P(u)$"	(by substitution)
"$\vdash P(u) \rightarrow P(t)$"	(by symmetry and substitution)

and returns

"$\vdash P(t) \leftrightarrow P(u)$"	(by definition of $\leftrightarrow$).

To test PRED_FCONV, we apply it to the depth conversion RD_CONV, from a previous session. The converted formula consists of the predicate P applied to the terms *abs*2 and *condfst*:

```
#PRED_FCONV RD_CONV "P (^abs2, ^condfst)";;
"⊢P((λt.(λu.t, u)FF)TT, (FST(TT, FF)⇒x | y)) ↔ P((TT, FF), x)" : thm.
```

PPLAMBDA's inference rules allow us to implement conversion operators for the quantifiers and logical connectives. The conversion SUB_FCONV applies a conversion

to all top-level terms and formulas of a formula:

```
let SUB_FCONV conv fconv =
    FIRST_FCONV [CONJ_FCONV fconv;
                 DISJ_FCONV fconv;
                 IMP_FCONV fconv;
                 IFF_FCONV fconv;
                 FORALL_FCONV fconv;
                 EXISTS_FCONV fconv;
                 PRED_FCONV conv];;
```

For mapping a conversion over all subformulas of a formula, the conversions DEPTH_FCONV, REDEPTH_FCONV, and TOP_DEPTH_FCONV are defined like their term analogs. For example:

```
letrec DEPTH_FCONV conv fconv fm =
      (SUB_FCONV conv (DEPTH_FCONV conv fconv) THENFC
       (REPEATFC fconv))
      fm;;
```

Let us bind some formula rewrites and test formulas for later sessions:

```
#let [P_Q; LESS_⊥] = example_frewrites;;
P_Q = "⊢∀x.P(x, x) ↔ Q x" : thm
LESS_⊥ = "⊢∀x.x ⊆ ⊥ ↔ x ≡ ⊥" : thm

#let [imp1; disj1; equiv1] = example_forms;;
imp1 =
  "∀x y.P((TT ⇒ y | z), SND(y, y)) → Q((λp.(p ⇒ v | y))FF)" : form
disj1 = "∃x.x ⊆ ⊥ TT ∨ SND(x, TT) ≡ ⊥ ⇒ TT | FF)" : form
equiv1 =
  "∀x.∃p.(FST(p, p) ⇒ x | ⊥) ≡ (p ⇒ (λz.SND(x, z))x | (λr.r) ⊥)"
  : form.
```

We make a conversion to use our formula rewrites, make a depth conversion from this and the term conversion RD_CONV, and try it on our test data:

```
#let MANY_FCONV =
     FIRST_FCONV (map REWRITE_FCONV [P_Q; LESS_⊥]);;
MANY_FCONV = - : fconv

#let D_FCONV = DEPTH_FCONV RD_CONV MANY_FCONV;;
D_FCONV = - : fconv

#D_FCONV imp1;;
"⊢(∀x y.P((TT ⇒ y | z), SND(y, y)) → Q((λp.(p ⇒ v | y))FF)) ↔
   (∀x y.Q y → Q y)"
: thm
```

```
#D_FCONV disj1;;
"⊢(∃x.x ⊆ ⊥ TT ∨ SND(x, TT) ≡ ⊥ ⇒ TT|FF)) ↔
   (∃x.x ≡ ⊥ ∨ TT ≡ ⊥)"
:thm

#D_FCONV equiv1;;
"⊢(∀x.∃p.(FST(p, p) ⇒ x|⊥) ≡ (p ⇒ (λz.SND(x, z))x|(λr.r)⊥)) ↔
   (∀x.∃p.(p ⇒ x|⊥) ≡ (p ⇒ x|⊥))"
:thm.
```

These formulas are not fully simplified. The next section shows how to eliminate subformulas, such as TT≡⊥, that are obviously true or false.

7.2. *Eliminating propositional tautologies*

LCF includes conversions that recognize propositional tautologies. For instance, TAUT_CONJ_FCONV can derive

```
TRUTH( ) ∧ A      ↔ A
A ∧ TRUTH( )      ↔ A
FALSITY( ) ∧ A    ↔ FALSITY( )
A ∧ FALSITY( )    ↔ FALSITY( ).
```

Most of the tautology conversion functions are hand-coded to treat a particular class of formulas. But the ones for quantifiers are implemented in terms of formula conversions. LCF has stored the theorems

```
FORALL_TRUTH      ⊢(∀x.TRUTH( ))      ↔ TRUTH( )
FORALL_FALSITY    ⊢(∀x.FALSITY( ))    ↔ FALSITY( ).
```

Using these, the 'forall' tautology conversion can simplify ∀x.TRUTH( ) and ∀x.FALSITY( ). Its ML definition is

```
let TAUT_FORALL_FCONV =
   (REWRITE_FCONV FORALL_TRUTH)
     ORELSEFC
   (REWRITE_FCONV FORALL_FALSITY);;
```

The family of conversions is modular. To improve the tautology test, write a better version of TAUT_FORALL_FCONV. Perhaps it should simplify ∀x.A to A for any formula A that does not contain x.

LCF provides the conversion BASIC_TAUT_FCONV, which tries all the tautology tests in turn, failing if none apply:

```
let BASIC_TAUT_FCONV =
   FIRST_FCONV [TAUT_CONJ_FCONV;
                TAUT_DISJ_FCONV;
                TAUT_IMP_FCONV;
```

```
TAUT_IFF_FCONV;
TAUT_FORALL_FCONV;
TAUT_EXISTS_FCONV;
TAUT_PRED_FCONV];;
```

There are many ways of building simplifiers from these conversions. The standard one, BASIC_FCONV, uses TOP_DEPTH_CONV to simplify terms, TOP_DEPTH_FCONV to simplify formulas, and BASIC_TAUT_FCONV to find tautologies in the resulting formulas. Many factors play a role in tailoring a simplifier to a specific problem. For instance, the REDEPTH conversions are slower but more thorough than the DEPTH ones. For current LCF applications, the TOP_DEPTH conversions seem to offer the best compromise of speed and generality.

```
let BASIC_FCONV conv fconv =
    TOP_DEPTH_FCONV (TOP_DEPTH_CONV conv)
      (fconv ORELSEFC BASIC_TAUT_FCONV);;
```

The following session shows how BASIC_FCONV combines our top-level conversions, MANY_CONV and MANY_FCONV. The resulting formula conversion solves the tautologies that were missed before:

```
#let B_FCONV = BASIC_FCONV MANY_CONV MANY_FCONV;
B_FCONV = - : fconv

# B_FCONV imp 1;;
"⊢(∀x y.P((TT ⇒ y | z), SND(y, y)) → Q((λp.(p ⇒ v | y)FF)) ↔
   TRUTH( )"
: thm

# B_FCONV disj 1;;
"⊢ (∃x.x ⊑ ⊥ TT ∨ SND(x, TT) ≡ (⊥ ⇒ TT | FF)) ↔
   (∃x.x ≡ ⊥)"
: thm
# B_FCONV equiv 1;;
"⊢(∀x.∃p. (FST(p, p) ⇒ x | ⊥) ≡ (p ⇒ (λz.SND(x, z))x | (λr.r) ⊥)) ↔
   TRUTH( )"
: thm.
```

8. Anatomy of a rewriting tactic

In studying these rewriting functions, let us remain aware of their original purpose: to prove theorems. Now we will study the LCF tactic REWRITE_TAC, which simplifies a goal by rewriting it and removing tautologies. This tactic is evolving over time. Though the version described here is not the latest, its structure illustrates the practical use of conversions and higher-order functions.

8.1. Implicative rewrites

The functions REWRITE_CONV and REWRITE_FCONV accept rewriting theorems of the form $\vdash t \equiv u$ or $\vdash A \leftrightarrow B$. However, there are many equivalences that only hold under certain conditions. Consider a theory of lists with strict CONS and MAP functions. The following theorem holds only by virtue of the antecedent forcing x to be defined in the equivalence:

$$\neg x \equiv \bot \rightarrow \text{MAP}\, f\, (\text{CONS}\, x\, l) \equiv \text{CONS}(f\, x)(\text{MAP}\, f\, l).$$

If x is $\bot$, l is NIL, and f is the constant function $\lambda y.\text{TT}$, then the equivalence does not hold:

$$\begin{aligned}
&\text{MAP}\ (\lambda y.\text{TT})\ (\text{CONS}\ \bot\ \text{NIL}) \equiv \text{CONS}\ ((\lambda y.\text{TT})\bot)\ (\text{MAP}\ (\lambda y.\text{TT})\ \text{NIL})\\
&\leftrightarrow \qquad \text{(by strictness of CONS, definition of MAP)}\\
&\text{MAP}\ (\lambda y.\text{TT})\ \bot \equiv \text{CONS}\ ((\lambda y.\text{TT})\ \bot)\ \text{NIL}\\
&\leftrightarrow \qquad \text{(by beta-conversion, strictness of MAP)}\\
&\bot \equiv \text{CONS TT NIL}\\
&\leftrightarrow \qquad \text{(by totality of CONS)}\\
&\text{FALSITY( )}.
\end{aligned}$$

In general, these *implicative rewrites* may depend on more than one antecedent. LCF presumes them to have the form, for non-negative n,

$$A_1 \rightarrow (\ldots (A_n \rightarrow t \equiv u) \ldots)$$
$$A_1 \rightarrow (\ldots (A_n \rightarrow (B \leftrightarrow C)) \ldots).$$

How can a conversion use such a theorem, given an instance t' of the left-hand term t? If it can prove the instances of the antecedents, $A'_1, \ldots, A'_n$, then, by Modus Ponens, it can return the theorem $\vdash t' \equiv u'$. How should it try to prove the antecedents? The simplifiers in both Edinburgh LCF and the Boyer/Moore Theorem Prover [2] solve antecedents by recursively invoking the simplifier. However, there is no need to commit ourselves; we can pass any proof tactic as an argument. Conversions that attempt to prove instances of the antecedents using a tactic *tac* are

IMP_REW_CONV *tac* "$\vdash A_1 \rightarrow (\ldots (A_n \rightarrow t \equiv u) \ldots)$"
IMP_REW_FCONV *tac* "$\vdash A_1 \rightarrow (\ldots (A_n \rightarrow (B \leftrightarrow C)) \ldots)$".

8.2. Backwards chaining

Although the latest version of REWRITE_TAC invokes itself to prove the antecedents of implicative rewrites, we will examine an earlier, simpler version. It uses *backwards chaining*—a proof search that resembles the execution of PPLAMBDA implications as a PROLOG program [5]. It is implemented using PART_FMATCH to match the consequent of an implication.

Let us see how backwards chaining can solve antecendents of implicative rewrites. Typically, an antecedent will require that a list l be defined: $\neg l \equiv \bot$. Consider a first-order theory of lists, with NIL, a strict CONS, and an infix operator APP to append

lists. The theory includes theorems asserting that these constants create defined lists:

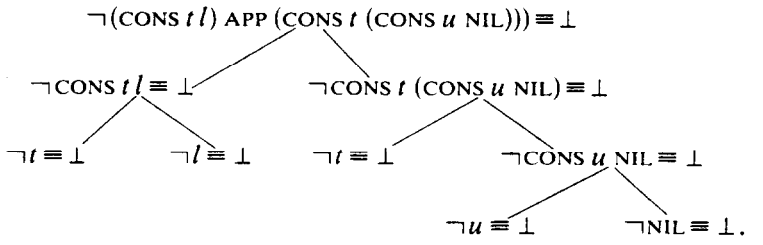
NIL_DEFINED $\vdash \neg \text{NIL} \equiv \bot$
CONS_DEFINED $\vdash \neg a \equiv \bot \rightarrow (\neg l \equiv \bot \rightarrow \neg \text{CONS}\ a\ l \equiv \bot)$
APP_DEFINED $\vdash \neg l_1 \equiv \bot \rightarrow (\neg l_2 \equiv \bot \rightarrow \neg l_1\ \text{APP}\ l_2 \equiv \bot)$.

The tactic IMP_SEARCH_TAC performs a depth-first search using a list of such theorems. It searches the list for a theorem whose consequent matches the goal, failing if there is none. Suppose there is a theorem

$$\vdash A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B,$$

and that the goal is an instance B' of B. The tactic calls itself recursively to prove the instances of the antecedents, $A_1' \ldots A_n'$, and proves the goal B' by Modus Ponens. For the search to terminate successfully, there must be theorems with no antecedents ($n = 0$). A theorem such as $\vdash A \rightarrow A$ will cause infinite regress.

In this example, it establishes that a complex list is defined, because it is constructed from the constants NIL, CONS, APP. The variables t, u, and l are all assumed to be defined. The goal tree shows how the initial goal is reduced to simpler subgoals, until the leaves are all true:



8.3. Canonical forms

A predicate logic such as PPLAMBDA allows many different ways of saying the same thing. For instance, $(A \wedge B) \rightarrow C$ is logically equivalent to $A \rightarrow (B \rightarrow C)$, though IMP_REW_CONV and IMP_SEARCH_TAC expect the latter. Edinburgh LCF addresses this problem by forcing *every* formula into a standard canonical form. Cambridge LCF, described here, takes a more flexible approach. It provides functions for putting theorems into canonical form; the user may invoke these or implement different ones in ML. These functions are *inference rules*. They do not simply manipulate data structures, but prove their output theorems from their input theorems.

The function IMP_CANON converts a theorem into a list of implications. This form is useful in many LCF situations. (Note: bound variables may be renamed; assumptions of the input theorem are passed to the output.)

IMP_CANON "$\vdash A \wedge B$" → (IMP_CANON "$\vdash A$")@(IMP_CANON "$\vdash B$")
IMP_CANON "$\vdash (\exists x.A) \rightarrow B$" → IMP_CANON "$\vdash A[y/x] \rightarrow B$"

IMP_CANON "$\vdash \forall x.A$" → IMP_CANON "$\vdash A[y/x]$"
IMP_CANON "$\vdash (A \wedge B) \to C$" → IMP_CANON "$\vdash A \to (B \to C)$"
IMP_CANON "$\vdash (A \vee B) \to C$" →
(IMP_CANON "$\vdash A \to C$") @ (IMP_CANON "$\vdash B \to C$").

If the cases above do not apply, and the argument is an implication $\vdash A \to B$, then IMP_CANON calls itself on the equivalent theorem, $A \vdash B$. This breaks B into a list $[A \vdash B_1; \ldots; A \vdash B_n]$ of theorems that assume A. It discharges A from each of these, to return $[A \to B_1; \ldots; A \to B_n]$. A theorem that satisfies no cases, such as $\vdash A \vee B$, passes through unchanged.

In this session, we see IMP_CANON converting several slightly different theorems into the same one:

```
#IMP_CANON (ASSUME "∀x. ¬x ≡ ⊥ → ∀y. ¬y ≡ ⊥ → ¬f x y ≡ ⊥");;
[·⊢" ¬x ≡ ⊥ → ¬y ≡ ⊥ → ¬f x y ≡ ⊥"]: thm list

#IMP_CANON (ASSUME "∀x y. ¬x ≡ ⊥ → ¬y ≡ ⊥ → ¬f x y ≡ ⊥");;
[·⊢" ¬x ≡ ⊥ → ¬y ≡ ⊥ → ¬f x y ≡ ⊥"]: thm list

#IMP_CANON (ASSUME "(¬x ≡ ⊥ ∧ ¬y ≡ ⊥) → ¬f x y ≡ ⊥");;
[·⊢" ¬x ≡ ⊥ → ¬y ≡ ⊥ → ¬f x y ≡ ⊥"]: thm list

#IMP_CANON (ASSUME "(¬x ≡ ⊥ ∧ ¬y ≡ ⊥) →
#                   ∀f g. ¬f x y ≡ ⊥ ∧ ¬g x y ≡ ⊥");;
[·⊢" ¬x ≡ ⊥ → ¬y ≡ ⊥ → ¬f x y ≡ ⊥";
 ·⊢" ¬x ≡ ⊥ → ¬y ≡ ⊥ → ¬g x y ≡ ⊥"]
: thm list.
```

Such implications are fine for IMP_REW_CONV and IMP_SEARCH_TAC, but IMP_REW_FCONV expects theorems of the uncommon form $C \leftrightarrow D$. The inference rule FCONV_CANON alters the consequent of certain implications into logical equivalences:

$P(x)$ → $P(x) \leftrightarrow$ TRUTH( ) (P a predicate symbol)
$\neg P(x)$ → $P(x) \leftrightarrow$ FALSITY( )
$C \leftrightarrow D$ → unchanged
else fail.

Thus FCONV_CANON proves logical equivalences that rewrite predicates to TRUTH( ) and negated predicates to FALSITY( ). It passes on any formula rewrites it encounters. For instance, the theorems NIL_DEFINED, CONS_DEFINED, and APP_DEFINED become implicative formula rewrites. (Recall that the formula $t \equiv u$ is shorthand for the predicate *equiv*(t, u).)

$\vdash$NIL $\equiv \bot \leftrightarrow$ FALSITY ( )
$\vdash \neg a \equiv \bot \to (\neg l \equiv \neg \to$ (CONS $a\ l \equiv \bot \leftrightarrow$ FALSITY( )))
$\vdash \neg l_1 \equiv \bot \to (\neg l_2 \equiv \bot \to (l_1$ APP $l_2 \equiv \bot \leftrightarrow$ FALSITY( ))).

These theorems can solve the antecedents of implicative rewrites using rewriting instead of backwards chaining. The effect is similar to the derivation tree shown at the end of the previous section. TOP_DEPTH_FCONV tries these formula rewrites *before* descending into the antecedent to apply term rewrites.

Thus, if the antecedent can be solved by backwards chaining, it can also be solved by rewriting, with comparable efficiency. Rewriting can solve many more antecedents than backwards chaining can. Its main drawback is a greater danger of infinite regress, trying to rewrite the antecedent of the antecedent of the .... Any implicative rewrite $\vdash A(t) \rightarrow t \equiv u$ can cause such regress, invoking itself in the attempt to prove its own antecedent. Again we see that many considerations guide the selection of components when building a simplifier.

8.4. *The primitive conversion tactic*

Though there are many ways to build a conversion, there is only one obvious way to reduce a goal using a conversion. The tactic (FCONV_TAC *fconv*) uses *fconv* to convert a goal A to a subgoal B, leaving its assumptions unchanged. If B is just TRUTH( ), then FCONV_TAC has achieved the goal A, and returns an empty subgoal list.

8.5. The rewriting tactic

The tactic REWRITE_TAC requires all the above pieces. It accepts a list of theorems, and puts them into canonical form using IMP_CANON and FCONV_CANON. It handles implicative rewrites using IMP_REW_CONV and IMP_REW_FCONV, which fail on unacceptable theorems. The function *mapfilter* gathers the successful conversions; these, along with BETA_CONV, are combined using FIRST_CONV, FIRST_FCONV, and BASIC_FCONV. REWRITE_TAC solves antecedents of implicative rewrites by backwards chaining, using the tactic IMP_SEARCH_TAC. To solve trivial subgoals in chaining, it augments the list of theorems with LCF's reflexivity axiom:

EQ_REFL $\vdash \forall x.x \equiv x$.

The tactic ASM_REWRITE_TAC calls REWRITE_TAC, using the tactical ASSUM_LIST to append the goal's assumptions to the input list of theorems. Most proofs rely on ASM_REWRITE_TAC for rewriting, in order to take advantage of the assumptions:

```
let REWRITE_TAC thl =
    let thms = flat (map IMP_CANON thl) in
    let chain_tac = IMP_SEARCH_TAC (EQ_REFL.thms) in
    let conv =
        FIRST_CONV (mapfilter (IMP_REW_CONV chain_tac) thms)
        ORELSEC BETA_CONV
    in
```

```
  let fconv =
      FIRST_FCONV
      (mapfilter ((IMP_REW_FCONV chain_tac)∘FCONV_CANON) thms)
    in
    FCONV_TAC (BASIC_FCONV conv fconv);;

let ASM_REWRITE_TAC thl =
    ASSUM_LIST (λasl.REWRITE_TAC (asl @ thl));;
```

9. Examples of solving goals by rewriting

To see ASM_REWRITE_TAC in use, consider a recent proof of mine [13]. It uses a data structure for expressions composed of constants, variables, and combinations of other expressions. It concerns infix functions OCCS and OCCS_EQ; these are relations because they return a truth-valued result. The relation "t OCCS u" searches u for an occurrence of t, returning TT if it finds one. The relation OCCS_EQ is the reflexive closure of OCCS. They are defined in terms of a boolean operator OR, and a computable equality relation =:

```
let OCCS_EQ =
new_axiom ('OCCS_EQ',
"∀t u.t OCCS_EQ u ≡ (t = u) OR (t OCCS u)");;

let OCCS_CLAUSES =
new_axiom ('OCCS_CLAUSES',
"∀t.t OCCS⊥ ≡ ⊥
  ∧
  (∀c. ¬c ≡ ⊥ →
    t OCCS (CONST c) ≡ FF)
  ∧
  (∀v. ¬v ≡ ⊥ →
    t OCCS (VAR v) ≡ FF)
  ∧
  (∀t1 t2. ¬t1 ≡ ⊥ → ¬t2 ≡ ⊥ →
    t OCCS (COMB t1 t2) ≡ (t OCCS_EQ t1) OR (t OCCS_EQ t2))");;
```

We will see how ASM_REWRITE_TAC helps to prove that the relation OCCS is transitive:

```
∀t. ¬t ≡ ⊥ →
  ∀u.t OCCS u ≡ TT →
    ∀u'.u OCCS u' ≡ TT → t OCCS u' ≡ TT.
```

Inducting on the variable u' yields four subgoals:

```
#expand (TERM_TAC "u");;
OK..
4 subgoals
"u OCCS (COMB t1 t2) ≡ TT → t OCCS (COMB t1 t2) ≡ TT"
  ["¬t ≡ ⊥"]
  ["t OCCS u ≡ TT"]
  ["u OCCS t1 ≡ TT → t OCCS t1 ≡ TT"]
  ["u OCCS t2 ≡ TT → t OCCS t2 ≡ TT"]
  ["¬t1 ≡ ⊥"]
  ["¬t2 ≡ ⊥"]

"u OCCS (VAR v) ≡ TT → t OCCS (VAR v) ≡ TT"
  ["¬t ≡ ⊥"]
  ["t OCCS u ≡ TT"]
  ["¬v ≡ ⊥"]

"u OCCS (CONST c) ≡ TT → t OCCS (CONST c) ≡ TT"
  ["¬t ≡ ⊥"]
  ["t OCCS u ≡ TT"]
  ["¬c ≡ ⊥"]

"u OCCS ⊥ ≡ TT → t OCCS ⊥ ≡ TT"
  ["¬t ≡ ⊥"]
  ["t OCCS u ≡ TT"].
```

Compare these with the axiom OCCS_CLAUSES; three of them contradict the antecedent, u OCCS $u' \equiv$ TT. Using the axioms OCCS_CLAUSES and OCCS_EQ, the tactic ASM_REWRITE_TAC solves the three easy goals. During initialization, it splits OCCS_CLAUSES into $\bot$, CONST, VAR, and COMB clauses, each an implicative rewrite. While rewriting the goal involving CONST, it notices the assumption $\neg c \equiv \bot$, and rewrites u OCCS (CONST c) to FF. Then it rewrites the antecedent FF $\equiv$ TT to FALSITY( ). Similarly, it rewrites the consequent to FALSITY( ), yielding the trivial goal FALSITY( ) → FALSITY( ). ASM_REWRITE_TAC solves the goals involving $\bot$ and VAR in the same way. The fourth goal is difficult, but the tactic advances it considerably:

```
"((u = t1) OR (u OCCS t1)) OR
 ((u = t2) OR (u OCCS t2)) ≡ TT
 →
 ((t = t1) OR (t OCCS t1)) OR
 ((t = t2) OR (t OCCS t2)) ≡ TT"
    ["¬t ≡ ⊥")
    ["t OCCS u ≡ TT"]
    ["u OCCS t1 ≡ TT → t OCCS t1 ≡ TT"]
    ["u OCCS t2 ≡ TT → t OCCS t2 ≡ TT"]
```

$$[\text{“}\neg t_1 \equiv \bot\text{”}]$$
$$[\text{“}\neg t_2 \equiv \bot\text{”}].$$

My paper [13] describes the rest of the proof, involving a case split followed by a further call to ASM_REWRTIE_TAC.

9.1. *Recent improvements to* REWRITE_TAC

REWRITE_TAC has evolved since this paper was first written. The version presented above is simpler than the latest one. Now we examine, in less detail, the recent improvements. The first implementation contained lots of messy code; I later realized how to express this in a modular style, using conversions.

The main innovation is to use *local assumptions* during rewriting. In the formulas $A \wedge B$ and $A \to B$, it is legitimate to assume A when rewriting B. Informally, the truth value of B is irrelevant unless A holds. A more convincing justification is the following derived inference rule, implemented in ML using the primitive rules:

$$\frac{\begin{array}{ll} & A \leftrightarrow A_2 \\ [A; A_2] & B \leftrightarrow B_2 \end{array}}{\begin{array}{l} (A \wedge B) \leftrightarrow (A_2 \wedge B_2) \\ (A \to B) \leftrightarrow (A_2 \to B_2). \end{array}}$$

This rule allows us to convert the formula A to A_2, then assume these while converting B to B_2, without these assumptions appearing in the resulting theorems.

Local assumptions facilitate many proofs. The older rewriting tactic could return a goal containing the formula

$$p \equiv \text{TT} \to p \Rightarrow x \mid y \equiv x.$$

Rewriting the conditional with the local assumption $p \equiv \text{TT}$ simplifies the formula to TRUTH( ). If the rewriting tactic cannot make local assumptions, then the user must manipulate the antecedent $p \equiv \text{TT}$ into a global position. These manipulations obscure the proof. They reflect the context of the formula, yet the formula is true in any context.

It is not obvious how the conversion (IMP_FCONV *fconv*) can make local assumptions. Somehow the assumption $p \equiv \text{TT}$ must be incorporated into *fconv*, itself built from other operators. My solution is to supply an additional argument, *fconv_fun*, which maps a formula to a conversion. The local version of IMP_FCONV is

LOCAL_IMP_FCONV *fconv fconv_fun*.

When converting an implication $A \to B$, it uses *fconv* to convert A to A_2, applies *fconv_fun* to A_2 to produce a new conversion, and converts B using that. Likewise there is a LOCAL_CONJ_FCONV. Combining these conversions with those for the other connectives yields a LOCAL_SUB_FCONV, a LOCAL_TOP_DEPTH_FCONV, and a LOCAL_BASIC_FCONV. These local conversions resemble their predecessors, but take the additional argument *fconv_fun*.

The new REWRITE_TAC is implemented in terms of a highly recursive formula conversion. For solving implicative rewrites, this conversion uses a call to itself rather than IMP_SEARCH_TAC. It passes another recursive call as the *fconv_fun* argument of LOCAL_SUB_FCONV. For fast pattern matching, it stores rewrites in *discrimination nets* [4] instead of lists.

A separate improvement is to expand out disjunctions during rewriting, causing automatic case splits. The conversion EXPAND_DISJ_FCONV derives

$$(A \vee B) \rightarrow C \leftrightarrow (A \rightarrow C) \wedge (B \rightarrow C)$$
$$(A \vee B) \wedge C \leftrightarrow (A \wedge C) \vee (B \wedge C)$$
$$C \wedge (A \vee B) \leftrightarrow (C \wedge A) \vee (C \wedge B).$$

This is applied at the same point as the tautology conversions. Existential quantifiers are similarly expanded. Expansion of disjunctions is especially effective when A and B, as local assumptions, help to rewrite C.

10. Conclusions

You may be thinking, "Conversions seem interesting, but must be hopelessly inefficient". Conversions are heavily used in Cambridge LCF, where they are efficient enough to prove difficult theorems. Runtime ranges from ten seconds to several minutes on a VAX 750 computer.

It is hard to improve the efficiency in the LCF framework. Any simplifier must produce a theorem to justify its result. It must coexist with other theorem-proving tools, and with ML. This precludes some optimizations, such as reducing the number of substitutions by maintaining a global environment of variable bindings [2]. The current ML compiler generates poor code, though efficient implementations are being developed [3].

Conversion functions have many advantages over Edinburgh LCF's simplifier, a large and inscrutable ML program. The operators PART_FMATCH, REWRITE_CONV, TAUT_CONJ_CONV, IMP_CANON, etc., carry out small, welldefined tasks. They have simple specifications and implementations. Together they express the rewriting tactic, REWRITE_TAC, in only a dozen lines.

REWRITE_TAC performs the vast majority of inferences in LCF proofs. Its limitations and abilities are easy to grasp, thanks to its modular structure. This helps the user to plan interactive sessions, and to read tactical proofs like a summary of the hundreds of formal inferences.

Conversions illustrate the power of higher-order functions. Because ML treats functions as first-class data, we can implement rewriting tools as functions and write operators to combine them. Proof tactics are functions too; the conversion IMP_REW_CONV generates and proves subgoals using a tactic passed to it as an argument. The instantiation function PART_TMATCH accepts a function argument that tells it what part of a theorem to match. This programming style differs greatly

from the style that Backus [1] recommends, where only first-order functions are allowed.

LCF is used not only for performing particular proofs, but also for research into proof techniques. The programming language ML provides the flexibility needed for this research. The discovery of the operators THEN, ORELSE, and REPEAT, for combining tactics, was a breakthrough in the development of Edinburgh LCF. So it is exciting to find similar operators for combining conversions, especially since tactics and conversions have little else in common. This calls for an inquiry into other instances of this programming style.

Acknowledgment

I would like to thank J. Fairbairn, M. Gordon, D. Matthews, and the referees for their detailed comments on the paper. D. Benson introduced me to semirings. This research is supported by the Science Research Council of Great Britain.

References

[1] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21** (1978) 613–641.
[2] R. Boyer and J. Moore, *A Computational Logic* (Academic Press, New York, 1979).
[3] L. Cardelli, The functional abstract machine, *Polymorphism: the ML/LCF/HOPE Newsletter* (Bell Laboratories, Murray Hill, NJ January 1983).
[4] E. Charniak, C. Riesbeck and D. McDermott, *Artificial Intelligence Programming* (Lawrence Erlbaum Associates, 1980).
[5] W. Clocksin and C. Mellish, *Programming in Prolog* (Springer, Berlin, 1981).
[6] A. Cohn, The equivalence of two semantic definitions: a case study in LCF, *SIAM. J. Comput.* **12** (1983) 267–285.
[7] M. Gordon, R. Milner and C. Wadsworth, *Edinburgh LCF* (Springer, Berlin, 1979).
[8] M. Gordon, Representing a logic in the LCF metalanguage, in: D. Neel, Ed., *Tools and Notions for Program Construction* (Cambridge University Press, Cambridge, 1982) 163–185.
[9] G. Huet and D. Oppen, Equations and rewrite rules: a survey, in: R. Book, Ed., *Formal Language Theory: Perspectives and Open Problems* (Academic Press, New York, 1980) 349–406.
[10] W. Kuechlin, An implementation and investigation of the Knuth–Bendix completion procedure, Report No. 17/82, Institut für Informatik I, University of Karlsruhe (1982).
[11] D.J. Lehmann, Algebraic structures for transitive closure, *Theoret. Comput. Sci.* **4** (1977) 59–76.
[12] Z. Manna, *Mathematical Theory of Computation* (McGraw-Hill, New York, 1974).
[13] L. Paulson, Recent developments in LCF: examples of structural induction, Report No. 34, Computer Laboratory, University of Cambridge (1983).
[14] L. Paulson, The revised logic PPLAMBDA: a reference manual, Report No. 36, Computer Laboratory, University of Cambridge (1983).